4. TestAnalysisandDesign–390minutes

Keywords

acceptance criteria, acceptance test-driven development, black-box test technique, boundary value analysis, branch coverage, checklist-based testing, collaboration-based test approach, coverage, coverage item, decision table testing, equivalence partitioning, error guessing, experience-based test technique,exploratorytesting,state transitiontesting,statementcoverage,testtechnique, white-boxtest technique

LearningObjectivesforChapter 4:

4.1 TestTechniquesOverview

FL-4.1.1 (K2)Distinguishblack-box,white-boxandexperience-basedtesttechniques

4.2 Black-boxTestTechniques

FL-4.2.1 (K3)Useequivalence partitioningtoderivetest cases

FL-4.2.2 (K3)Useboundaryvalueanalysistoderivetestcases FL-

4.2.3 (K3) Use decision table testing to derive test cases

FL-4.2.4 (K3) Use state transition testing to derive test cases

4.3 White-boxTest Techniques

FL-4.3.1 (K2)Explainstatementtesting

FL-4.3.2 (K2) Explain branch testing

FL-4.3.3 (K2)Explainthevalueofwhite-boxtesting

4.4 Experience-basedTestTechniques

FL-4.4.1 (K2)Explainerrorguessing

FL-4.4.2 (K2)Explainexploratorytesting

FL-4.4.3 (K2)Explainchecklist-basedtesting

4.5. Collaboration-basedTestApproaches

FL-4.5.1 (K2)Explainhow towriteuserstoriesincollaborationwithdevelopersandbusiness representatives

FL-4.5.2 (K2)Classifythedifferentoptionsforwritingacceptancecriteria

FL-4.5.3 (K3)Useacceptancetest-drivendevelopment(ATDD)toderivetestcases

4.1. TestTechniquesOverview

Test techniques support the tester in test analysis (what to test) and in test design (how to test). Test techniques helptodevelopa relativelysmall,butsufficient,setoftestcasesinasystematicway. Test techniques alsohelpthe testertodefinetestconditions,identifycoverageitems, andidentifytestdata during the test analysis and design. Further information on test techniques and their corresponding measures can be found in the ISO/IEC/IEEE 29119-4 standard, and in (Beizer 1990, Craig 2002, Copeland 2004, Koomen 2006, Jorgensen 2014, Ammann 2016, Forgács 2019).

Inthissyllabus,testtechniquesareclassifiedasblack-box,white-box,andexperience-based.

Black-boxtesttechniques(alsoknownasspecification-basedtechniques)arebasedonananalysisof the specified behavior of the test object without reference to its internal structure. Therefore, the test cases are independent of how the software is implemented. Consequently, if the implementation changes, but the required behavior stays the same, then the test cases are still useful.

White-boxtesttechniques(alsoknownasstructure-basedtechniques) arebasedonananalysis ofthe test object's internal structure and processing. As the test cases are dependent on how the software is designed, they can only be created after the design or implementation of the test object.

Experience-based test techniques effectively use the knowledge and experience of testers for the designandimplementationoftestcases. Theeffectiveness ofthese techniquesdependsheavily onthe tester'sskills.Experience-based testtechniquescandetectdefectsthatmaybe missedusingtheblack- box andwhite-box testtechniques. Hence,experience-basedtesttechniques arecomplementary tothe black-box and white-box test techniques.

4.2. Black-BoxTestTechniques

Commonlyusedblack-boxtesttechniquesdiscussedinthefollowingsectionsare:

- EquivalencePartitioning
- BoundaryValueAnalysis
- DecisionTableTesting
- StateTransition Testing

4.2.1. Equivalence Partitioning

Equivalence Partitioning (EP) divides data into partitions (known as equivalence partitions) based on the expectation that all the elements of a given partition are to be processed in the same way by the test object. The theory behind this technique is that if a test case, that tests one value from an equivalence partition, detects adefect,thisdefect should alsobedetectedbytestcases thattestany other valuefrom the same partition. Therefore, one test for each partition is sufficient.

Equivalence partitions can be identified for any data element related to the test object, including inputs, outputs,configuration items,internalvalues, time-related values,andinterfaceparameters.Thepartitions maybecontinuousor discrete,orderedor unordered,finiteorinfinite.Thepartitions mustnotoverlapand must be non-empty sets.

Forsimpletest objectsEPcan beeasy,butinpractice,understanding howthetestobjectwilltreat different values is often complicated. Therefore, partitioning should be done with care.

A partition containing valid values is called a valid partition. A partition containing invalid values is called an invalid partition. The definitions of valid and invalid values may vary among teams and organizations. For example, valid values may be interpreted as those that should be processed by the test object or as thoseforwhichthespecificationdefinestheirprocessing. Invalidvalues maybeinterpretedas those that should be ignoredorrejectedby thetest object or as thoseforwhich noprocessingis definedinthetest object specification.

In EP, the coverage items are the equivalence partitions. To achieve 100% coverage with this technique, test cases must exercise all identified partitions (including invalid partitions) by covering each partition at leastonce. Coverageismeasuredas thenumberof partitionsexercisedbyatleastone testcase, divided by the total number of identified partitions, and is expressed as a percentage.

Many test objects include multiple sets of partitions (e.g., test objects with more than one input parameter), which means that a test case will cover partitions from different sets of partitions. The simplest coverage criterion in the case of multiple sets of partitions is called Each Choice coverage (Ammann 2016).EachChoicecoverage requirestestcasestoexerciseeachpartitionfromeachset of partitions at least once. Each Choice coverage does not take into account combinations of partitions.

4.2.2. BoundaryValue Analysis

Boundary Value Analysis (BVA) is a technique based on exercising the boundaries of equivalence partitions. Therefore,BVAcanonly beusedfororderedpartitions. The minimumand maximumvaluesof a partition are its boundary values. In the case of BVA, if two elements belong to the same partition, all elements between them must also belong to that partition.

BVAfocusesontheboundaryvaluesof thepartitionsbecausedevelopersaremorelikelytomakeerrors with these boundary values. Typical defects found by BVA are located where implemented boundaries are misplaced to positions above or below their intended positions or are omitted altogether.

Thissyllabuscoverstwo versionsoftheBVA:2-value and3-valueBVA.Theydifferinterms ofcoverage items per boundary that need to be exercised to achieve 100% coverage.

In 2-value BVA (Craig 2002, Myers 2011), for each boundary value there are two coverage items: this boundaryvalueand itsclosestneighborbelongingtothe adjacent partition.Toachieve100%coverage with 2-value BVA, test cases must exercise all coverage items, i.e., all identified boundary values. Coverageismeasuredasthenumberofboundaryvalues thatwereexercised,dividedby thetotal number of identified boundary values, and is expressed as a percentage.

In3-valueBVA(Koomen 2006, O'Regan 2019),foreach boundaryvaluetherearethreecoverageitems: this boundary value and both its neighbors. Therefore, in 3-value BVA some of the coverage items may not be boundary values. To achieve 100% coverage with 3-value BVA, test cases must exercise all coverage items, i.e., identified boundary values and their neighbors. Coverage is measured as the number of boundary values and their neighbors exercised, divided by the total number of identified boundary values and their neighbors, and is expressed as a percentage.

3-valueBVAismorerigorousthan 2-value BVAasitmay detectdefects overlookedby 2-valueBVA.For example, if the decision "if (x ≤ 10) …" is incorrectly implemented as "if (x = 10) …", no test data derived from the 2-value BVA (x = 10, x = 11) can detect the defect. However, x = 9, derived from the 3-value BVA, is likely to detect it.

4.2.3. DecisionTableTesting

Decisiontables areusedfortestingtheimplementationof systemrequirementsthatspecifyhow different combinationsofconditions resultindifferent outcomes.Decisiontables are aneffectiveway ofrecording complex logic, such as business rules.

When creating decision tables, the conditions and the resulting actions of the system are defined. These formtherows ofthe table. Eachcolumncorrespondsto a decisionrulethatdefinesauniquecombination of conditions, along with the associated actions. In limited-entry decision tables all the values of the conditions andactions (exceptforirrelevantorinfeasible ones; see below) areshownas Booleanvalues (true orfalse).Alternatively,in extended-entry decisiontables some orall theconditionsandactionsmay also take on multiple values (e.g., ranges of numbers, equivalence partitions, discrete values).

Thenotation forconditions isasfollows:“T”(true)means thattheconditionissatisfied.“F”(false)means that the condition is not satisfied. “–” means that the value of the condition is irrelevant for the action outcome. “N/A” means that the condition is infeasible for a given rule. For actions: “X” means that the action should occur. Blank means that the action should not occur. Other notations may also be used.

A full decision table has enough columns to cover every combination of conditions. The table can be simplified by deleting columns containing infeasible combinations of conditions. The table can also be minimizedby mergingcolumns,in whichsomeconditions donot affecttheoutcome,intoasinglecolumn. Decision table minimization algorithms are out of scope of this syllabus.

In decision table testing, the coverage items are the columns containing feasible combinations of conditions.Toachieve100%coveragewiththistechnique,testcasesmust exerciseallthesecolumns. Coverage is measured as the number of exercised columns, divided by the total number of feasible columns, and is expressed as a percentage.

The strength of decision table testing is that it provides a systematic approach to identify all the combinations of conditions, some of which might otherwise be overlooked. It also helps to find any gaps or contradictions in the requirements. If there are many conditions, exercising all the decision rules may betimeconsuming,since thenumberof rulesgrowsexponentiallywiththenumber ofconditions. Insuch a case, to reduce the number of rules that need to be exercised, a minimized decision table or a risk-based approach may be used.

4.2.4. StateTransitionTesting

A statetransition diagrammodels the behaviorof asystemby showingits possiblestates andvalid state transitions. A transition is initiated by an event, which may be additionally qualified by a guard condition. Thetransitions areassumedtobeinstantaneousandmaysometimes result inthesoftwaretakingaction. The common transition labeling syntax is as follows: “event [guard condition] / action”. Guard conditions and actions can be omitted if they do not exist or are irrelevant for the tester.

A state table is a model equivalent to a state transition diagram. Its rows represent states, and its columns represent events (together with guard conditions if they exist). Table entries (cells) represent transitions, andcontain thetarget state, aswellastheresultingactions,ifdefined.Incontrasttothestate transition diagram, the state table explicitly shows invalid transitions, which are represented by empty cells.

A test case based on a state transition diagram or state table is usually represented as a sequence of events,which resultsinasequence ofstatechanges (andactions,if needed). Onetestcase may,and usually will, cover several transitions between states.

Thereexistmanycoveragecriteriaforstatetransitiontesting.Thissyllabusdiscussesthreeof them.

In **all states coverage**, the coverage items are the states. To achieve 100% all states coverage, test casesmust ensurethatallthe states arevisited.Coverageismeasured asthe numberofvisitedstates divided by the total number of states, and is expressed as a percentage.

In **valid transitions coverage** (also called 0-switch coverage), the coverage items are single valid transitions. Toachieve100%validtransitions coverage,testcasesmustexerciseallthevalidtransitions. Coverage is measured as the number of exercised valid transitions divided by the total number of valid transitions, and is expressed as a percentage.

In**alltransitionscoverage**, thecoverageitems are allthetransitionsshown inastatetable.Toachieve 100% all transitions coverage, test cases must exercise all the valid transitions and attempt to execute invalid transitions. Testing only one invalid transition in a single test case helps to avoid fault masking, i.e., a situation in which one defect prevents the detection of another. Coverage is measured as the number of valid and invalid transitions exercised or attempted to be covered by executed test cases, divided by the total number of valid and invalid transitions, and is expressed as a percentage.

Allstatescoverageisweakerthanvalid transitionscoverage,becauseitcantypicallybeachievedwithout exercising all the transitions. Valid transitions coverage is the most widely used coverage criterion. Achievingfullvalidtransitionscoverageguaranteesfullall statescoverage. Achievingfullalltransitions coverage guarantees both full all states coverage and full valid transitions coverage and should be a minimum requirement for mission and safety-critical software.

4.3. White-BoxTest Techniques

Becauseoftheirpopularityand simplicity,thissectionfocusesontwo code-relatedwhite-boxtest techniques:

- Statementtesting
- Branchtesting

There are morerigorous techniquesthatareused insomesafety-critical,mission-critical, orhigh-integrity environmentstoachievemorethoroughcodecoverage. There are alsowhite-boxtesttechniques usedin higher test levels (e.g., API testing), or using coverage not related to code (e.g., neuron coverage in neural network testing). These techniques are not discussed in this syllabus.

4.3.1. StatementTestingandStatementCoverage

In statement testing, the coverage items are executable statements. The aim is to design test cases that exercisestatementsinthecodeuntilanacceptablelevel of coverageisachieved.Coverageismeasured as the number of statements exercised by the test cases divided by the total number of executable statements in the code, and is expressed as a percentage.

When 100% statement coverage is achieved, it ensures that all executable statements in the code have beenexercised atleast once. Inparticular,this meansthateachstatementwitha defect will beexecuted, which may cause a failure demonstrating the presence of the defect. However, exercising a statement with a test case will not detect defects in all cases. For example, it may not detect defects that are data dependent (e.g., a division by zero that only fails when a denominator is set to zero). Also, 100% statementcoverage does notensure thatallthedecisionlogichasbeentested as,for instance,it may not exercise all the branches (see chapter 4.3.2) in the code.

4.3.2. BranchTestingandBranchCoverage

A branch is a transfer of control between two nodes in the control flow graph, which shows the possible sequences inwhichsourcecodestatementsareexecutedinthetestobject. Eachtransferofcontrolcan be either unconditional (i.e., straight-line code) or conditional (i.e., a decision outcome).

In branch testing the coverage items are branches and the aim is to design test cases to exercise branchesinthecode untilan acceptablelevelofcoverageisachieved.Coverage is measuredas the number of branches exercised by the test cases divided by the total number of branches, and is expressed as a percentage.

When 100% branch coverage is achieved, all branches in the code, unconditional and conditional, are exercised by test cases. Conditional branches typically correspond to a true or false outcome from an "if...then" decision, an outcome from a switch/case statement, or a decision to exit or continue in a loop. However,exercisinga branchwith atestcasewillnotdetectdefects inallcases.Forexample,it maynot detect defects requiring the execution of a specific path in a code.

Branchcoveragesubsumesstatementcoverage.Thismeansthatanysetoftestcasesachieving100% branch coverage also achieves 100% statement coverage (but not vice versa).

4.3.3. The ValueofWhite-boxTesting

A fundamental strength that all white-box techniques share is that the entire software implementation is takenintoaccountduringtesting,whichfacilitates defectdetectionevenwhenthesoftwarespecification is vague, outdated or incomplete. A corresponding weakness is that if the software does not implement one or more requirements, white box testing may not detect the resulting defects of omission (Watson 1996).

White-boxtechniquescan beusedinstatictesting (e.g., during dryrunsofcode).Theyarewellsuited to reviewing code that is not yet ready for execution (Hetzel 1988), as well as pseudocode and other high-level or top-down logic which can be modeled with a control flow graph.

Performing only black-box testing does not provide a measure of actual code coverage. White-box coverage measures provide an objective measurement of coverage and provide the necessary informationto allowadditionalteststobegeneratedtoincreasethiscoverage, andsubsequentlyincrease confidence in the code.

4.4. Experience-basedTestTechniques

Commonlyusedexperience-basedtesttechniquesdiscussedinthefollowingsectionsare:

- Errorguessing
- Exploratorytesting
- Checklist-basedtesting

4.4.1. ErrorGuessing

Errorguessingisatechniqueusedtoanticipatetheoccurrenceoferrors,defects, andfailures, basedon the tester's knowledge, including:

- Howtheapplicationhasworkedinthepast

- Thetypesoferrorsthe developerstendtomakeand thetypesofdefects thatresultfromthese errors
- Thetypesoffailuresthathaveoccurredinother,similarapplications

In general, errors, defects and failures may be related to: input (e.g., correct input not accepted, parameterswrongormissing),output(e.g.,wrongformat,wrongresult), logic(e.g.,missingcases, wrong operator), computation (e.g., incorrect operand, wrong computation), interfaces (e.g., parameter mismatch, incompatible types), or data (e.g., incorrect initialization, wrong type).

Faultattacksareamethodicalapproachtotheimplementationoferrorguessing.Thistechniquerequires the tester to create or acquire a list of possible errors, defects and failures, and to design tests that will identify defects associated with the errors, expose the defects, or cause the failures. These lists can be built based on experience, defect and failure data, or from common knowledge about why software fails.

See(Whittaker2002,Whittaker2003,Andrews2006)formoreinformationonerrorguessing and fault attacks.

4.4.2. Exploratory Testing

Inexploratorytesting,testsaresimultaneouslydesigned,executed,andevaluatedwhilethetesterlearns about the test object. The testing is used to learn more about the test object, to explore it more deeply with focused tests, and to create tests for untested areas.

Exploratory testing is sometimes conducted using session-based testing to structure the testing. In a session-based approach, exploratory testing is conducted within a defined time-box. The tester uses a test charter containing test objectives to guide the testing. The test session is usually followed by a debriefingthatinvolves adiscussion betweenthe testerandstakeholders interestedin thetestresultsof the test session. In this approach test objectives may be treated as high-level test conditions. Coverage items are identified and exercised during the test session. The tester may use test session sheets to document the steps followed and the discoveries made.

Exploratory testing is useful when there are few or inadequate specifications or there is significant time pressure on the testing. Exploratory testing is also useful to complement other more formal test techniques.Exploratorytestingwillbemoreeffectiveifthetesterisexperienced,has domain knowledge and has a high degree of essential skills, like analytical skills, curiosity and creativeness (see section 1.5.1).

Exploratory testingcanincorporatetheuseofothertesttechniques (e.g.,equivalencepartitioning). More information about exploratory testing can be found in (Kaner 1999, Whittaker 2009, Hendrickson 2013).

4.4.3. Checklist-BasedTesting

In checklist-based testing, a tester designs, implements, and executes tests to cover test conditions from achecklist.Checklistscanbe builtbasedonexperience, knowledge about whatisimportantforthe user, or an understanding of why and how software fails. Checklists should not contain items that can be checkedautomatically,items bettersuitedasentry/exitcriteria,oritems thataretoogeneral (Brykczynski1999).

Checklist items are often phrased in the form of a question. It should be possible to check each item separately and directly. These items may refer to requirements, graphical interface properties, quality characteristicsor otherformsoftestconditions.Checklistscanbecreatedtosupportvarioustest types, including functional and non-functional testing (e.g., 10 heuristics for usability testing (Nielsen 1994)).

Somechecklistentries maygraduallybecomelesseffectiveovertimebecausethedeveloperswilllearn to avoid making the same errors. New entries may also need to be added to reflect newly found high severity defects. Therefore, checklists should be regularly updated based on defect analysis. However, care should be taken to avoid letting the checklist become too long (Gawande 2009).

Intheabsence ofdetailedtestcases, checklist-basedtestingcanprovide guidelinesand some degreeof consistencyforthe testing.Ifthechecklistsarehigh-level, somevariabilityintheactualtestingislikelyto occur, resulting in potentially greater coverage but less repeatability.

4.5.Collaboration-basedTestApproaches

Each of the above-mentioned techniques (see sections 4.2, 4.3, 4.4) has a particular objective with respecttodefectdetection.Collaboration-basedapproaches,ontheotherhand, focus alsoondefect avoidance by collaboration and communication.

4.5.1. CollaborativeUserStoryWriting

Auserstory represents afeaturethatwillbevaluableto eitherauserorpurchaserof asystem or software. User stories have three critical aspects (Jeffries 2000), called together the "3 C's":

- Card–themedium describingauserstory(e.g.,anindexcard,anentryinanelectronic board)
- Conversation–explainshowthesoftware willbeused(canbedocumented orverbal)
- Confirmation–theacceptancecriteria(seesection4.5.2)

Themostcommonformatfor auserstoryis"Asa[role],I want[goal tobe accomplished],sothatI can [resulting business value for the role]", followed by the acceptance criteria.

Collaborativeauthorshipoftheuserstorycanusetechniquessuchas brainstormingandmindmapping. The collaboration allows the team to obtain a shared vision of what should be delivered, by taking into account three perspectives: business, development and testing.

Good user stories should be: Independent, Negotiable, Valuable, Estimable, Small and Testable (INVEST).Ifastakeholderdoesnot know how to test auserstory, this mayindicate that theuserstoryis notclear enough,orthatitdoesnot reflectsomethingvaluableto them, orthatthestakeholderjustneeds help in testing (Wake 2003).

4.5.2. AcceptanceCriteria

Acceptancecriteriafora userstoryaretheconditionsthat animplementationof the userstorymustmeet to be accepted by stakeholders. From this perspective, acceptance criteria may be viewed as the test conditions that should be exercised by the tests. Acceptance criteria are usually a result of the Conversation (see section 4.5.1).

Acceptancecriteriaareusedto:

- Definethescope ofthe userstory
- Reachconsensusamongthestakeholders
- Describebothpositiveandnegativescenarios

- Serveasabasisfor the userstoryacceptancetesting(seesection4.5.3)

- Allowaccurateplanningandestimation

Thereare severalwaystowriteacceptance criteriaforauserstory.Thetwomostcommon formats are:

- Scenario-oriented(e.g.,Given/When/ThenformatusedinBDD,seesection2.1.3)
- Rule-oriented(e.g.,bulletpointverificationlist,ortabulatedformofinput-output mapping)

Mostacceptancecriteriacanbedocumentedin oneofthesetwoformats. However, theteam mayuse another, custom format, as long as the acceptance criteria are well-defined and unambiguous.

4.5.3. AcceptanceTest-drivenDevelopment(ATDD)

ATDDisatest-firstapproach (see section 2.1.3).Testcasesarecreatedpriortoimplementing theuser story. The test cases are created by team members with different perspectives, e.g., customers, developers, and testers (Adzic 2009). Test cases may be executed manually or automated.

The first step is a specification workshop where the user story and (if not yet defined) its acceptance criteria are analyzed, discussed, and written by the team members. Incompleteness, ambiguities, or defectsintheuserstoryareresolvedduringthisprocess. Thenextstep istocreatethetest cases. This can be done by the team as a whole or by the tester individually. The test cases are based on the acceptance criteria and can be seen as examples of how the software works. This will help the team implement the user story correctly.

Sinceexamples andtestsarethesame, thesetermsare oftenusedinterchangeably. Duringthetest design the test techniques described in sections 4.2, 4.3 and 4.4 may be applied.

Typically, the first test cases are positive, confirming the correct behavior without exceptions or error conditions, andcomprisingthesequenceofactivitiesexecutedifeverythinggoes asexpected. Afterthe positive test cases are done, the team should perform negative testing. Finally, the team should cover non-functional qualitycharacteristics aswell (e.g., performance efficiency,usability). Test casesshould be expressed in a way that is understandable for the stakeholders. Typically, test cases contain sentences in natural language involving the necessary preconditions (if any), the inputs, and the postconditions.

The test cases must cover all the characteristics of the user story and should not go beyond the story. However,theacceptancecriteriamay detailsomeof theissuesdescribedin theuserstory. Inaddition, no two test cases should describe the same characteristics of the user story.

Whencapturedinaformatsupported byatestautomationframework,thedeveloperscan automatethe test cases by writing the supporting code as they implement the feature described by a user story. The acceptance tests then become executable requirements.